Name _____

GPU Microarchitecture
EE 7722
Solve-Home Final Examination

Wednesday, 28 April 2021 to Saturday, 1 May 2021 16:00 CDT

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(50 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck! Help Keep Everyone Safe!

Problem 1: [50 pts] The motivation for Homework 2 was the frustratingly slow execution of our loop nest on Nvidia CC 7.5 devices. For each floating-point operation a thread needed to do two loads. Since loads issue on 1/4 the rate of 32-bit multiply/add instructions, our rate of execution is seemingly $1/(1+4+4) = \frac{1}{9}$ that of the floating point bandwidth for the device.

Appearing in this problem are runs of kernels based on the solution to Homework 2. It would be a good idea to look over the solution, it has been checked into the repo and the main file is at `https://www.ece.lsu.edu/gp/2021/hw02-sol.cu.html`. In particular, look at `dnn_sol_b`, which is similar to the kernel discussed here, `dnn_fe`.

Unlike `dnn_sol_b`, kernel `dnn_fe`, and has template parameters for the blocking factors `bn`, `bo`, and `bm`. These template parameters make it easier to do runs comparing different blocking factors. A shortened version of this kernel appears below. The full version is in the repo in directory `hw/gpm/2021/fe` and an htmlized version of the code is at `https://www.ece.lsu.edu/gp/2021/fe.cu.html`.

```
template<int nn=0, int nc=0, int ni=0, int bn=8, int bo=2, int bm=4>
__global__ void dnn_fe(Layer l) {
  const int tid = blockIdx.x * blockDim.x + threadIdx.x;
  const int num_threads = blockDim.x * gridDim.x;
  const int no = ni,  nm = nc;

  acc_t* const ai = (acc_t*) __builtin_assume_aligned(l.ai_d,16);
  acc_t* const ao = (acc_t*) __builtin_assume_aligned(l.ao_d,16);
  wht_t* const w = (wht_t*) __builtin_assume_aligned(l.w_d,16);

  constexpr int ab = bo * bn * bm;
  const int nnmo = nn * nm * no,  nnmo_ab = nnmo / ab,  nn_bn = nn / bn;
  const int nm_bm = nm / bm;

  for ( int inmo = tid; inmo < nnmo_ab; inmo += num_threads ) {
      const int im_bm = inmo % nm_bm,   im0 = im_bm * bm,   ino = inmo / nm_bm;
      const int in_bn = ino % nn_bn,    in0 = in_bn * bn,  io_bo = ino / nn_bn;
      const int io0 = io_bo * bo;

      acc_t ac[bo][bm][bn]{}; // Storage for outputs computed here.
      for ( int ic = 0; ic < nc; ic++ )
#pragma unroll 4
        for ( int ii = 0; ii < ni; ii++ ) {

            // Pre-load the inputs (ai) that will be needed, one input
            // for each of bn batches. Each will be used bm * bo times.
            acc_t ain[bn];
            for ( int i_bn = 0; i_bn < bn; i_bn++ )  {
                const int in = in0 + i_bn;
                ain[i_bn] = ai[ ii + ni * ( ic + nc * in ) ]; }

            for ( int i_bm = 0; i_bm < bm; i_bm++ )
              for ( int i_bo = 0; i_bo < bo; i_bo++ ) {
                  const int im = im0 + i_bm,   io = io0 + i_bo;
                  // Load a weight. The weight will be used bn times.
                  wht_t wht = w[ im + nm * ( ii + ni * ( ic + nc * io ) ) ];
                  for ( int i_bn = 0; i_bn < bn; i_bn++ )
                    ac[i_bo][i_bm][i_bn] += ain[i_bn] * wht;  }}

      for ( int i_bn = 0; i_bn < bn; i_bn++ )
        for ( int i_bm = 0; i_bm < bm; i_bm++ )
          for ( int i_bo = 0; i_bo < bo; i_bo++ ) {
              const int io = io0 + i_bo,  in = in0 + i_bn,  im = im0 + i_bm;
              ao[ io + no * ( im + nm * in ) ] = ac[i_bo][i_bm][i_bn]; }
    }
}
```

A run of the final exam code launches the `dnn_base` and multiple versions of `dnn_fe` each with different block sizes and different blocking factors. The network layer itself is run in two sizes.

In the program output the blocking factors are shown to the right of the kernel, just above the table of outputs. The label `out/iter` is a product of the blocking factors.

As with the homework assignment, a bar graph is shown which indicates how closely each run is to each of three limiters: off-chip data bandwidth (`*`), FP bandwidth (`+`), and instruction issue bandwidth (`-`). Use the FP bandwidth plot to gauge performance. (See the Homework 2 handout for more information on the graph.)

All runs were performed on an RTX 2080 Super. Key data appears below.

```
GPU 0: NVIDIA GeForce RTX 2080 SUPER @ 1.81 GHz WITH 7982 MiB GLOBAL MEM
GPU 0: L2: 4096 kiB   MEM<->L2: 496.1 GB/s
GPU 0: CC: 7.5  MP: 48  CC/MP:  64  DP/MP:  2  TH/BL: 1024
GPU 0: SHARED: 49152 B/BL  65536 B/MP  CONST: 65536 B  # REGS: 65536
GPU 0: PEAK: 5576 SP GFLOPS  174 DP GFLOPS  COMP/COMM:  45.0 SP  2.8 DP
```

(*a*) Appearing below is the output of `dnn_fe` run at two different blocking factors.

```
(dnn_fe<ls[1].nn,ls[1].nc,ls[1].ni,4,1,1>)   bn=4, bo=1, bm=1, out/iter=4
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 52 48  1  2.1  2.4  52.6  10550  118 +------
200 52 48  2  2.1  2.5  40.8   5754  217 ++------------
200 52 48  3  2.1  3.0  28.1   5113  244 ++-------------
200 52 48  4  2.1  2.8  22.7   3967  314 *+------------------
200 52 48  8  2.1  2.0  13.5   2177  572 *+++--------------------------------
200 52 48 12  2.1  1.7  10.5   1608  775 *++++-------------------------------->
200 52 48 16  2.1  1.5   9.3   1630  765 *++++-------------------------------->
(dnn_fe<ls[1].nn,ls[1].nc,ls[1].ni,2,1,2>)   bn=2, bo=1, bm=2, out/iter=4
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 52 48  1  2.0  2.5  71.7  10314  121 +-------
200 52 48  2  2.0  2.5  38.2   5343  233 ++-------------
200 52 48  3  2.0  2.6  27.1   4702  265 ++---------------
200 52 48  4  2.0  2.8  21.5   3897  320 *+------------------
200 52 48  8  2.0  2.0  13.1   2128  586 *+++---------------------------------
200 52 48 12  2.0  1.7  10.5   1479  843 **++++--------------------------------->
200 52 48 16  2.0  1.5   9.3   1501  830 **++++--------------------------------->
```

With both blocking factors 4 outputs will be computed for each `inmo` iteration. Notice that the performance of the second configuration, where `bn=2,bo=1,bm=2` is faster than the `bn=4,bo=1,bm=1` configuration. Explain why that might be so and compute, based on device characteristics, the difference in performance between these two configurations.

☐ Reason that `bn=2,bo=1,bm=2` slightly faster.

☐ Compute exactly how much faster `bn=2,bo=1,bm=2` should be based on this reason.

(*b*) Appearing below are execution of the kernels with two different blockings. Pay attention to what happens as the number of warps per SM increases from 1 to 4. For this device there are four warp schedulers per SM, so we expect the device to be underutilized when there are fewer than 4 warps. Data from the `bn=4` kernel is consistent with that, as are the kernels computing fewer outputs per iteration. However, the `bn=8` kernel shows no benefit on a launch with four warps over a launch with three warps.

```
(dnn_fe<ls[1].nn,ls[1].nc,ls[1].ni,4,2,4>)   bn=4, bo=2, bm=4, out/iter=32
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 52 48  1  1.2  2.1  19.4   6027  207 +------------
200 52 48  2  1.2  1.6  11.1   3311  376 *++----------------------
200 52 48  3  1.2  1.4   8.3   2266  550 *+++---------------------------------
200 52 48  4  1.2  1.3   6.9   1715  726 *++++------------------------------------>
200 52 48  8  1.2  1.2   4.9   1198 1040 *******-------------------------------->
200 52 48 12  1.2  1.1   4.3    729 1710 ************---------------------------->
(dnn_fe<ls[1].nn,ls[1].nc,ls[1].ni,8,2,4>)   bn=8, bo=2, bm=4, out/iter=64
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 52 48  1  1.1  1.6  11.2   2822  442 *++-------------------------
200 52 48  2  1.1  1.3   7.0   1457  855 *******-------------------------------->
200 52 48  3  1.1  1.2   5.6    972 1282 *********------------------------------>
200 52 48  4  1.1  1.2   4.9    975 1278 *********------------------------------>
200 52 48  8  1.1  1.1   4.1    650 1916 ************---------------------------->
200 52 48 12  1.1  1.1   3.8    714 1746 ************---------------------------->
```

☐ Explain why the performance at 3 and 4 warps is the same for the `bn=8` kernel.

Hint: Look at utilization.

(*c*) The runs below are on the smaller networks. Notice that too much blocking seems to hurt performance.

```
Launching with 48 blocks of up to 32 warps.
(dnn_base<ls[0].nn,ls[0].nc,ls[0].ni>)   bn=1, bo=1, bm=1, out/iter=1
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 20 32  1  3.3  0.9  53.9    662  124 +-------
200 20 32  2  3.3  0.3  43.3    363  226 *+------------
200 20 32  3  3.3  0.3  33.6    284  288 *+----------------
200 20 32  4  3.3  0.3  27.2    254  322 *+------------------
200 20 32  8  3.3  0.3  17.8    228  359 *++--------------------
200 20 32 12  3.3  0.2  14.7    218  376 *++----------------------
200 20 32 16  3.3  0.2  13.2    233  351 *+--------------------
200 20 32 20  3.3  0.2  12.1    238  344 *+--------------------
200 20 32 24  3.3  0.2  11.7    225  364 *++--------------------
200 20 32 28  3.3  0.2  11.2    216  379 *++----------------------
200 20 32 32  3.3  0.2  11.0    246  334 *+--------------------
(dnn_fe<ls[0].nn,ls[0].nc,ls[0].ni,4,1,1>)   bn=4, bo=1, bm=1, out/iter=4
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 20 32  1  2.1  0.2  29.4    600  137 +--------
200 20 32  2  2.1  0.2  18.8    335  244 *+-------------
200 20 32  3  2.1  0.2  15.3    246  332 *+------------------
200 20 32  4  2.1  0.2  13.6    213  384 *++----------------------
200 20 32  8  2.1  0.2  10.9    130  630 **++------------------------------------>
200 20 32 12  2.1  0.2  10.1    114  720 **++++----------------------------------->
200 20 32 16  2.1  0.2   9.7    129  633 **++-----------------------------------> 
200 20 32 20  2.1  0.3   9.3    158  517 **++------------------------------
200 20 32 24  2.1  0.2   9.5    111  738 **++++----------------------------------->
200 20 32 28  2.1  0.3   9.5    128  638 **++-----------------------------------> 
200 20 32 32  2.1  0.3   9.5    145  567 **++---------------------------------
(dnn_fe<ls[0].nn,ls[0].nc,ls[0].ni,2,1,2>)   bn=2, bo=1, bm=2, out/iter=4
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 20 32  1  2.0  0.2  28.0    564  145 +--------
200 20 32  2  2.0  0.2  18.1    305  268 *+---------------
200 20 32  3  2.0  0.2  14.9    215  382 *++----------------------
200 20 32  4  2.0  0.2  13.2    180  456 *++--------------------------
200 20 32  8  2.0  0.2  10.8    118  693 **++++----------------------------------->
200 20 32 12  2.0  0.2  10.0    103  795 **++++----------------------------------->
200 20 32 16  2.0  0.2   9.6    116  709 **++++----------------------------------->
200 20 32 20  2.0  0.3   9.4    142  575 **++-------------------------------
200 20 32 24  2.0  0.2   9.4    100  818 **++++----------------------------------->
200 20 32 28  2.0  0.3   9.5    117  702 **++++----------------------------------->
200 20 32 32  2.0  0.3   9.6    131  624 **++-----------------------------------
(dnn_fe<ls[0].nn,ls[0].nc,ls[0].ni,4,1,2>)   bn=4, bo=1, bm=2, out/iter=8
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 20 32  1  1.6  0.2  18.8    353  232 *+-------------
200 20 32  2  1.6  0.2  13.6    215  380 *++----------------------
200 20 32  3  1.6  0.2  11.8    157  520 **++-----------------------------
200 20 32  4  1.6  0.2  10.9    122  669 **++++----------------------------------->
200 20 32  8  1.6  0.2   9.7    107  764 **++++----------------------------------->
200 20 32 12  1.6  0.2   9.6     83  982 **+++++---------------------------------->
200 20 32 16  1.6  0.2   9.4    110  748 **++++----------------------------------->
200 20 32 20  1.6  0.3   8.1    135  607 **++-----------------------------------
200 20 32 24  1.6  0.3   8.8    109  753 **++++----------------------------------->
200 20 32 28  1.6  0.3  10.4    128  641 **++-----------------------------------> 
```

```
200 20 32 32  1.6  0.3  11.8    147  556 *+++-------------------------------
(dnn_fe<ls[0].nn,ls[0].nc,ls[0].ni,4,1,4>)   bn=4, bo=1, bm=4, out/iter=16
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 20 32  1  1.3  0.2  13.6    253  324 *+-------------------
200 20 32  2  1.3  0.2  10.9    139  588 **++-----------------------------------
200 20 32  3  1.3  0.2  10.1    101  808 **++++------------------------------------>
200 20 32  4  1.3  0.2   9.8    107  766 **++++------------------------------------>
200 20 32  8  1.3  0.2   9.5     91  896 **++++------------------------------------>
200 20 32 12  1.3  0.3   7.1     89  920 **++++------------------------------------>
200 20 32 16  1.3  0.3   7.6    116  708 **++++------------------------------------>
200 20 32 20  1.3  0.2  10.6    148  555 *+++-------------------------------
200 20 32 24  1.3  0.2  12.5    175  467 *++--------------------------
200 20 32 28  1.3  0.2  13.6    202  405 *++-------------------------
200 20 32 32  1.3  0.2  14.4    229  358 *++-----------------------
(dnn_fe<ls[0].nn,ls[0].nc,ls[0].ni,4,2,4>)   bn=4, bo=2, bm=4, out/iter=32
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 20 32  1  1.2  0.1   9.3    284  288 *+-----------------
200 20 32  2  1.2  0.2   6.9    218  376 *++----------------------
200 20 32  3  1.2  0.2   6.1    125  654 **++++------------------------------------>
200 20 32  4  1.2  0.2   5.8    127  644 **++++------------------------------------>
200 20 32  8  1.2  0.3   5.4    138  592 **++-----------------------------------
200 20 32 12  1.2  0.3   4.6    154  531 *++-------------------------------
200 20 32 16  1.2  0.3   4.4    179  458 *++--------------------------
200 20 32 20  1.2  0.3   4.0    206  398 *++-----------------------
200 20 32 24  1.2  1.3   3.0    246  333 *+--------------------
(dnn_fe<ls[0].nn,ls[0].nc,ls[0].ni,8,2,4>)   bn=8, bo=2, bm=4, out/iter=64
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 20 32  1  1.1  0.3   7.1    192  426 *++-------------------------
200 20 32  2  1.1  0.2   5.9    118  695 **++++------------------------------------>
200 20 32  3  1.1  0.3   5.7    116  704 **++++------------------------------------>
200 20 32  4  1.1  0.3   5.4    119  691 **++++------------------------------------>
200 20 32  8  1.1  0.3   3.6    151  542 *++++-------------------------------
200 20 32 12  1.1  0.3   3.1    190  432 *++-------------------------
```

☐ Try to determine the amount of blocking (values of `bn`, `bo`, `bm`) that will produce the best performance in terms of device and layer characteristics.

(*d*) Notice that the higher `out/iter` the lower the maximum number of warps per MP. That can be seen in the runs below.

```
(dnn_fe<ls[1].nn,ls[1].nc,ls[1].ni,4,1,1>)   bn=4, bo=1, bm=1, out/iter=4
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 52 48  1  2.1  2.4  52.6  10550  118 +------
200 52 48  2  2.1  2.5  40.8   5754  217 ++------------
200 52 48  3  2.1  3.0  28.1   5113  244 ++-------------
200 52 48  4  2.1  2.8  22.7   3967  314 *+------------------
200 52 48  8  2.1  2.0  13.5   2177  572 *+++-----------------------------------
200 52 48 12  2.1  1.7  10.5   1608  775 *++++----------------------------------->
200 52 48 16  2.1  1.5   9.3   1630  765 *++++----------------------------------->
200 52 48 20  2.1  1.4   8.5   1631  764 *++++----------------------------------->
200 52 48 24  2.1  1.4   7.9   1544  807 *+++++---------------------------------->
200 52 48 28  2.1  1.4   8.7   1451  859 **+++++--------------------------------->
200 52 48 32  2.1  1.7  11.0   1621  769 *++++----------------------------------->
(dnn_fe<ls[1].nn,ls[1].nc,ls[1].ni,8,2,4>)   bn=8, bo=2, bm=4, out/iter=64
 nn nc ni wp I/op DUse  2Use   t/s FP  === Util: FP++  Insn-- Data**  ========
200 52 48  1  1.1  1.6  11.2   2822  442 *++-------------------------
200 52 48  2  1.1  1.3   7.0   1457  855 **+++++---------------------------------->
200 52 48  3  1.1  1.2   5.6    972 1282 **++++++++------------------------------>
200 52 48  4  1.1  1.2   4.9    975 1278 **++++++++------------------------------>
200 52 48  8  1.1  1.1   4.1    650 1916 ***++++++++++-------------------------->
200 52 48 12  1.1  1.1   3.8    714 1746 ***++++++++++-------------------------->
```

☐ Why does increasing `out/iter` reduce the maximum number of warps per MP?

(*e*) The runs performed for Homework 2 and for this exam were all compute-bound on the RTX 2080 supper. Start with the large configuration, `nn=200`, `nc=nm=52`, `ni=no=48` and the blocking `bn=4`, `bo=1`, and `bm=4`. Describe a similar (but not identical) configuration in which the data limit and FP limit are identical.

☐ Configuration where off-chip data limit and FP limit are the same.

Problem 2: [20 pts] A goal of homework 2 was to take advantage of re-use to reduce the number of load instructions executed by each thread. A value would be loaded once and used multiple times.

Consider the following alternative methods of reducing the number of load instructions.

(*a*) Each thread loads one value and places it into shared memory, so the entire warp loads 32 values. Each thread uses all 32 values. Of course, assume that this is done correctly and that no thread is performing redundant calculations. (Something like this was done in `mxv_sh_ochunk` in file `vtx-xform-size.cu`, see `https://www.ece.lsu.edu/gp/2021/vtx-xform-size.cu.html`.) Does this solve our problem? Base your answer on the characteristics of shared memory instructions.

☐ Explain whether using shared memory solves the problem we were trying to solve in Homework 2.

☐ Estimate the improvement or worsening of the time needed to issue instructions.

(*b*) Each thread loads one value, and as in the previous sub-problem, all threads in the warp will need it. Rather than using shared memory, the threads use warp shuffle instructions. An execution of `nval = _shfl_sync(~0,myVal,lane);` will write `nval` with the `myVal` from the thread in lane `lane`. (See the documentation for CUDA intrinsic `__shfl_sync()`.) Does this solve our problem? Base your answer on documented characteristics of the warp shuffle. (For example, see the Performance Guidelines chapter of the CUDA Programming guide.)

☐ Explain whether shuffle instructions solves the problem we were trying to solve in Homework 2.

☐ Estimate the improvement or worsening of the time needed to issue instructions.

Problem 3: [30 pts] As has been pointed out in class, the seven-level loop nest used to describe computation in CNN layers is fairly simple and one might expect that a straightforward textbook procedure could be used to design an optimal CNN accelerator. But no. Many papers describing CNN accelerators claim an optimal design, but only optimal within a constrained set of possibilities. Yang [1] tries to look at a broader set of possible designs and draw conclusions about which accelerator hardware design or accelerator software design (dataflow, blocking) is best.

(*a*) Consider representing the CUDA GPU model using Yang's representation. Important issues to consider are storage levels, computation elements (by which I mean PEs, MACs, and stuff on a GPU), and communication. To understand CUDA communication issues think about ways in which one thread can obtain a value computed by another. How that's done depends upon the two threads' thread and block indices.

First, what CUDA construct should be represented as a PE (using Yang's definition)?

☐ Which would be best represented as a PE: a thread, a warp, a warp scheduler, a functional unit, or an SM? Explain.

(*b*) Yang uses $A|B$ to describe an assignment of work to a 2-D array of PEs such that the horizontal position (or column number or x-axis value) indicates a value of A (say the left most PE uses $a = 0$, the next one $a = 1$, etc) and the vertical position indicates a value of B. This is intended for CNN (or TPU) accelerators in which the PEs are arranged in a 2-D array and in which communication takes advantage of this arrangement. Suppose the thread index of a CUDA thread were used as the horizontal position and the block index were used as the vertical position. So for a $A|B$ dataflow `a=threadIdx.x` and `b=blockIdx.x`.

Is that a good way of representing CUDA threads in Yang's model? If not explain what about the performance of the CUDA code would not match the performance as modeled by these schemes (making needed adjustments to model the GPU). Propose a better way of mapping threads, warps, and blocks into these representations.

☐ Is it a good idea to use thread index for the horizontal position and block index for the vertical position?
☐ Explain. ☐ Describe any problems with this representation.

☐ Propose an alternative mapping, one which provides 2-D PE-array-like communication. (Remember that we are describing how the CUDA model of a GPU is represented, we are not changing the GPU itself.)

(*c*) Listing 2 in Yang shows how a blocking of the loop relates to storage levels. An important part of their analyses is determining how many times each level is read and written.

CUDA shared memory, as we all know, is private to a block. Can Yang's representation handle this? If so, how (other than making a block the equivalent of a PE)?

☐ Can, and if so how can, per-block storage be represented in Yang's representation?

References:

[1] Yang, X., Gao, M., Liu, Q., Setter, J., Pu, J., Nayak, A., Bell, S., Cao, K., Ha, H., Raina, P., Kozyrakis, C., and Horowitz, M. Interstellar: Using Halide's scheduling language to analyze DNN accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS 20, Association for Computing Machinery, p. 369383. https://doi.org/10.1145/3373376.3378514.